Generating Step Ordering
Experiments with CMAX

Clayton T. Morrison
Paul R. Cohen
USC Information Sciences Institute

30 April 2007 - Site Visit - BBN Tech
Outline

• The Role of CMAX in POIROT
• Year 1 experiments: step ordering
  – Producer/Consumer and Method Analysis
  – Generating Informative Tests
  – Planning Efficient Experiments
• Year 1 Final Goal, Progress
Causal MAp eXperiment designer
>> CMAX called as a service <<

Outline of Protocol for Interaction with POIROT components

- **Input:**
  - Background knowledge (resource model, experiment schemas, primitive operators, confirmed methods, causal fragments)
  - Hypotheses to analyze and/or test (one or more methods, prior trace(s), explicitly competing hypotheses)

- **Output:**
  - Experiment package

- **Input:**
  - Experiment Result

- **Output:**
  - Result Analysis: further experiments / suggested change

---

USC/ISI - CMAX - Morrison & Cohen 3
Step Ordering

- What is the problem we’re trying to solve in experimenting with *step ordering*?
  - The overall goal is to improve generalization.
  - Given a total linear order that solves the problem (i.e., the steps in the expert trace)...
  ... and a set of ordering constraints (asserted in hypothesized methods or identified in a step link analysis)...
  ... determine whether *other* (latent, hidden) ordering relationships present in the original total order are *necessary*.
Steps in Automating Design of Ordering Experiments

• Characterize the hypothesis space
  – Producer/Consumer and Method LTML analyses
• Efficiently generating space of informative tests
  – POC Permutation Tree
• Planning efficient and effective experiments
  – Test Relationship Graph
WIT Hypothesized Method

(Workflow witPlan
  (body
    (let ((links reqs - med@requirementSet) ...)
      (body
        (seq ;; seq1
          (step s1
            (activity lookupRequirements)
            (put (lookupRequirementsOut => reqs)))
          (step loop1
            (loop
              (links (req (over reqs) - med@PatientReqRecord))
              (body
                (let
                  (links ...)
                  (body
                    (seq ;; seq2
                      (step s2 ...)
                      ...
                      (step s11 ...))
                  (branch
                    (test (= missionOutcome res@Success))
                    (seq
                      (step s13 ...)
                      ...
                      (step s15...))
                    (test (NOT (= missionOutcome res@Success))
                      (choice
                        (step s12
                          (get (apoe <= apoe)
                            (apod <= apod)
                            (patient <= req)
                            (avTime <= arr-at-apoe))
                          (activity witMethod1))
                        (step s13
                          (get (patientID <= pID))
                          (activity reserveSeat))))
                  )))
            )))
      )))
  )))

Initial step

loop

branch

sequence (details on next slide...)

Based on Trace 2a - 16 Feb 07
(seq ; seq2
  (step s2
    (values)
    (asgn (p := (propval req med@patientID))
      (origin := (propval req loc@origin))
      (hosp := (propval req loc@destination))
      (LAT := (propval req trans@LAT))))

(step s3
  (get (locationID <= origin) (trans@radius <= 100))
  (activity lookupAirport)
  (put (LUAirportOut => airports1)))

(step s4
  (values)
  (asgn (apoe := (propval (nth 1 airports1) trans@airportLocationID))))

(step s5
  (get (patientID <= p) (locationID <= apoe))
  (activity setPatientAPOE))

(step s6
  (get (patientID <= p)
      (fromLocation <= origin) (toLocation <=
        (activity getArrivalTime)
        (put (departArrive => departArrive1)))

(step s66
  (values)
  (asgn (arr-at-apoe := (propval departArrive1 trans@arivalTime))))

(step s7
  (get (patientID <= p) (time <= arr-at-apoe))
  (activity setPatientAvailable))

(step s8
  (get (loc@locationID <= hosp) (trans@radius <=
      (activity lookupAirport)
      (put (LUAirportOut => airports2)))

(step s9
  (values)
  (asgn (apod := (propval (nth 1 airports2) trans@airportLocationID))))

(step s10
  (get (patientID <= p) (locationID <= apod))
  (activity setPatientAPOD))

(step s11
  (get (fromLocationID <= apoe) (toLocationID <= apod)
PC-Analysis on interval 2-8

What PC-Analysis is for
Characterize the space of possible causal relations
Factor the hyp space
Varying degree of constraint

Original Total Order
(2345678)

PC-matched order constraints (green)
((2 3) (2 4) (2 5) (2 8) (4 5) (4 8) (6 7) (6 8))

“Implicit” partial orders (non-links)
((2 6) (2 7) (3 4) (3 5) (3 6) (3 7) (3 8)
(4 6) (4 7) (5 6) (5 7) (5 8) (7 8))
What is a Test?

• A test is a workflow: a totally ordered sequence of steps
  – This workflow can be executed by SHOPPER
• The total order must satisfy every ordering constraint we wish to preserve, but may violate one or more of the other candidate ordering constraints.
• Asymmetry in outcome: If test workflow…
  – succeeds: then every violated hypothesized ordering constraint is not necessary
  – fails: failure might be because one or more of the violated hypothesized ordering constraints is necessary.
What are we testing?

- **Total Order**
- **Transitive Closure**
- **Ordering constraints (partial order)**
- **Trans closure of order constraints**
- **Implicit ordering constraints**
Generating Permutations Respecting Order Constraints - Expensive

# Violations | Run-lengths | Hypothesized OCs violated
---|---|---

| V | 1 | 2 | 3 | 4 | 5 | Re-Order | (2 6) | (2 7) | (3 4) | (3 5) | (3 6) | (3 7) | (3 8) | (4 6) | (4 7) | (5 6) | (5 7) | (5 8) | (7 8) |
| 12 | 1 | 0 | 0 | 0 | 0 | (6 7 2 4 8 5 3) | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 11 | 2 | 0 | 0 | 0 | 0 | (6 7 2 4 5 8 3) | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 11 | 1 | 0 | 0 | 0 | 0 | (6 7 2 4 8 5 3) | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 11 | 0 | 0 | 0 | 0 | 0 | (6 2 4 8 7 5 3) | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 11 | 0 | 0 | 0 | 0 | 0 | (6 2 7 4 8 5 3) | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 10 | 2 | 0 | 0 | 0 | 0 | (6 7 2 4 5 3 8) | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |
| 10 | 1 | 0 | 0 | 0 | 0 | (6 7 2 4 8 5 3) | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 10 | 1 | 0 | 0 | 0 | 0 | (6 7 2 4 3 8 5) | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |
| 10 | 1 | 0 | 0 | 0 | 0 | (6 7 2 4 7 8 5 3) | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 10 | 1 | 0 | 0 | 0 | 0 | (6 7 2 4 8 5 3) | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 10 | 0 | 0 | 0 | 0 | 0 | (6 2 4 8 7 5 3) | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 10 | 0 | 0 | 0 | 0 | 0 | (6 2 4 8 5 7 3) | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 10 | 0 | 0 | 0 | 0 | 0 | (6 2 4 7 8 5 3) | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 10 | 0 | 0 | 0 | 0 | 0 | (6 2 7 4 8 5 3) | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 9 | 3 | 1 | 0 | 0 | 0 | (6 7 2 4 8 5 3) | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 9 | 2 | 0 | 0 | 0 | 0 | (6 2 7 4 5 8 3) | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 9 | 1 | 0 | 0 | 0 | 0 | (6 2 7 4 8 5 3) | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

7! = 5040 Possible Permutations

144 Total orders respecting ordering constraints

0, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, ...
Generating Permutations with Order Constraints - more efficient

Swap step 0 to after step 5
Planning Experiments

• Given PO Permutation Tree, now select order of experiments
• Experiment generation policy depends on expected number of real implicit partial order constraints:
  – Few: Binary Search
    • 1 experiment to verify no hidden constraints: violate all!
    • $O(\log n)$ to verify exactly 1 hidden constraint
    • $O(2n)$ worst case (with many hidden constraints)
  – Many: Linear Search
    • $O(n)$
Test Dependency Graph

Example:

Ordering Constraints:
To Test

(0 1) (0 2) (0 3) (1 2) (2 3)

For “linear” test policy

1. Execute tests with minimal number of violations (initially row 1)
2. Update graph based on outcome
3. Repeat

For “binary” test policy

1. Execute tests with maximum set cover violations (greedy within log n of optimal)
2. Update graph based on outcome
3. Repeat
Conclusion

- Basic CMAX with ordering experiments integrated by end Year 1 (mid May)
  - Meta-learner interaction protocol
  - Parsing general LTML
  - Identifying sequence chunks
  - Core CMAX functionality
    - Step 1: PC Analysis
    - Step 2: Nearly-complete Test construction
    - Step 3: Experiment generation
  - Output workflow - (executable by SHOPPER)
  - Output Method change suggestion
The Role of CMAX in POIROT

• Overall Goal:
  – Target any expression or assertion in LTML as the subject of experiments to verify correct generalization
  – Minimize number of experiments (**efficient**)
  – Make experiments informative (**effective**)
  – Rate experiments according to how informative they are relative to a learning goal, their importance

• Classes of method constructs
  – Step ordering and dependencies
  – Branching conditions
  – Loops

• Year 1: **Step Ordering**
Producer/Consumer Analysis

Prior producers

<table>
<thead>
<tr>
<th>matched</th>
<th>unmatched</th>
</tr>
</thead>
</table>

Subsequent consumers

<table>
<thead>
<tr>
<th>matched</th>
<th>unmatched</th>
</tr>
</thead>
</table>

Analysis Interval

Given
(a) Total order, valid for a problem instance
(b) Primitive Operators

What PC-Analysis is for
Characterize the space of possible causal relations
Factor the hyp space
Varying degree of constraint

Types of PC-Analysis
1. Exact Value Match
2. Match Value Type
3. Value Functional Relationships
   (e.g., \( a \leq b \))
Trade-off Between Linear & Binary Search - 1 hidden OC